Information Systems Institute

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2011 (184.260)

Assignment 2

Submission Deadline: 11.5.2012, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the TUWEL forum) are allowed, but the code has to be written alone. If we find that two students submitted the same, or very similar assignments, these students will be graded with 0 points (no questions asked). We will use automated plagiarism checks to compare solutions. Note that we will also include the submissions of previous years in our plagiarism checks.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution includes all the libraries and dependencies and it compiles without errors. If you are unsure, test to compile your submission on a different computer (e.g., one of the ZID lab computers). **We are known to deduct points drastically if your code does not build/run out of the box.**
- Before solving the tasks concerning Enterprise Java Beans, we recommend reading Part IV of the Java EE 6 Tutorial[1]. If you are already familiar with EJB 3.0, this link[2] may give you a good impression of the new features in EJB 3.1 (find parts 1-4 of this series linked in the reference section there). However, note this has been published as a preview: not every feature mentioned made it into the final specification.
- You should use Java 1.6.14+[3], MySQL 5.1[4] as your database management system and GlassFish v3[5] as your application server. Task 1 explains how to set up Glassfish in detail.
- For your solution, use the provided project stubs:

  All needed libraries are already included if you want to use another one, feel free to integrate it. We expect (as you can see in the persistence-unit configuration of persistence.xml) that you setup a database dst that can be accessed by root (without a password). You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Build scripts for ant[6] are already included, you may modify them, but the submitted solution has to be compilable and runnable with the predefined targets.

[1] http://download.oracle.com/javaee/6/tutorial/doc/bnblr.html
[2] http://www.theserverside.com/news/1363649/New-Features-in-EJB-31-Part-5
[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[4] http://dev.mysql.com/downloads/mysql/5.1.html#downloads
[5] http://glassfish.java.net/downloads/v3-final.html
[6] http://ant.apache.org/

A. Code Part

1. Enterprise Java Beans 3.0 (26 Points)

Your task is to develop an enterprise application for the grid management system introduced in Assignment 1. Reuse your entity classes from task 1 (basic mapping and inheritance). Code from any other tasks is no longer required.

Firstly, install GlassFish and set GLASSFISH_HOME to the '**glassfish**'-directory right inside (!) the directory you installed GlassFish to. Start the application server by calling '**asadmin start-domain**' from the **bin**-directory, and, before calling '**ant setup**' from our build-file, make sure the file '**dst-ds.xml**' in the setup-directory is configured correctly. After a restart ('**asadmin restart-domain**') everything should be configured correctly. Also take a look at the GlassFish administration console at http://localhost:4848. By default, GlassFish uses the following file for logging (which may be important for debugging your solution):

- **GLASSFISH_HOME/domains/domain1/logs/server.log**

Please study the build file (**server/build.xml**) to understand the deployment process and complete the dist target, such that the classes the client needs (and only those) are included within the client library. In the client build file (**client/build.xml**) you have to complete the run target.

Put the server code into the **1_ejb/server** project, the client code into the **1_ejb/client** project. Make reasonable decisions concerning the type of session bean (stateful, stateless, singleton) to use for each task, and whether the beans should be remotely or locally available. Follow the principles of minimal visibility and minimal accessibility.

1.a. Session beans (15 Points)

- **Create a bean for testing purpose (TestingBean)**:

  Provide a method that inserts at least two grids, two users, two memberships, two clusters, five computers and one job, whose execution has started 30 minutes ago and has not finished yet (end field is set to null). Obviously, storing a grid may require to store some other entities as well. To test your solutions of the following tasks you may come back to this bean at a later point to store additional data. This method may be invoked by your client directly.

- **Create a bean for managing prices (PriceManagementBean)**:

  Our system requires a manageable pricing model for the charges of using the Grid. We assume that the user has to pay two different payments for every assigned job. As a sort of down payment, the user first pays a small fee when assigning a job. The purpose of this fee from the Grid provider's viewpoint is to hedge against the risk of non-payment. Hence, new users have to pay more than long-term users who have already proven to be reliable. In particular, this fee decreases with increasing number of jobs a user has submitted (and paid) in the past. For the second fee, the user's account is debited with the variable price for the job's execution time after the job has finished.

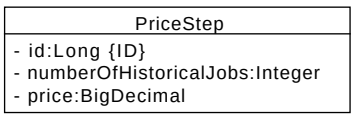

Figure 1: PriceStep Entity

The PriceManagerBean is a helper to adminster the "steps" of the price curve. This bean provides a method to store the price steps in the database, according to the number of jobs the user has

previously executed. (e.g.: 30€ for less than 100 executed jobs, 15€ for 100-1000 jobs, 5€ for 1000-5000 jobs ...). The *PriceStep* entity that should be stored is depicted in Figure 1.

The PriceManagementBean should also offer a method to get the fee for a given number of executed jobs. This bean only serves as a helper for the beans you will implement next. All prices should be retrieved from the database once the application server initializes the application and after that stored in memory (to avoid time-consuming database reads at runtime). At this point, creating the required persistent entity should be straightforward. The concrete implementation of the bean should be designed by you. In the end, the bean has to provide a possibility to set prices for a number of historical jobs and get the fee for a specified number of historical jobs.

Note that especially the last method (retrieving fees) may get called quite often, so you should keep performance in mind and think about the default behavior of concurrency and transactions managed by the container. However, as a change of prices is not expected to happen all that often, you can directly store new values in the database (do not forget to update your in-memory data structures, which you have loaded at application server startup, though!).

- **Create a bean for general management concerns (GeneralManagementBean)**:

  For now, this bean only has to provide a way to set prices using the bean you just created. We will extend this bean at a later point. However, note that all methods the bean will provide will share no state and will be invoked independently of each other. This bean should also be invokable by the client directly.

- **Create a bean that allows users to assign jobs for several grids (JobManagementBean)**:

  First of all this bean provides a method for the user to login with username and password.

  For more convenience the system provides the possibility to assign several jobs in a single transaction. To do this the user can add jobs for a single grid to a temporary job list by specifying the id of the grid, the number of CPUs, the workflow and parameters (as list) of the job. Think of the temporary job list as a 'shopping card' for jobs: users add jobs to the list one after another (possibly over a longer timespan) and submit them all in one go.

  When the bean receives requests to add jobs to the temporary job list, it first has to check if there are enough free computers for this grid left. To do this check if the sum of the CPUs of all free computers in the grid is larger than (or equal to) the CPUs necessary for the job. If this is not the case the job cannot be scheduled now, and the user is informed (throw a meaningful custom exception). If it is possible to execute the job the bean has to assign it to concrete computers. In this assignment we will ignore complex scheduling issues. You can simply query all free computers from the database and start assigning free computers at random until the sum of CPUs of all assigned computers is equal to (or larger than) the number of CPUs required by the job.

  After the list has been submitted to the system, all jobs in the list are started immediately (set the executions start field to now and the status to `SCHEDULED` – it is not possible to submit jobs where the scheduling starts in the future). When submitting, make sure to check again if the jobs can still be scheduled the way you planned (since the system has many users it is possible that another user was scheduling jobs in parallel using the same computers). Think of a suitable protocol to achieve this. If you find out that it is not possible anymore to execute the job, notify the user with an exception, as above.

  You should also provide a way to remove all jobs for a specific grid from the temporary job list, as well as a method to get the current amount of jobs for each grid in this list.

  Note that no data is written to the database before the temporary job list is finally submitted. So provide a method to do the final submission. Before the final submission, the user has to login at some point in the conversation. When the assignments are finally submitted, store the data (jobs and executions) to the database only if all chosen computers are available. Make sure this method executes transactionally secure, so that no data is written to the database if something goes wrong (i.e.: computers are not available). If the submission was successful, discard the bean. Otherwise,

throw meaningful exceptions so that the user can react to this and modify the job assignment (in case of an exception the bean should not be discarded!).

1.b. Timer Service and asynchronous method invocations (7 Points)

- **Implement a timer service used for simulation**

  Up to this point we only assigned jobs to the grid. Now, in order to test our solution, we need to simulate that jobs are actually going to finish at some point. For this we use a timer service. The timer service periodically (for instance every minute) takes the jobs that are running (start is set but end is null) and completes them (i.e., set the status to `FINISHED`, set the end date, and so on).

- **Provide a method to retrieve the bill of a user**

  Now you should extend the GeneralManagementBean that we have started earlier. Implement a method which can be used to retrieve the total bill of a user (given by username). That means that for each of this user's finished but unpaid jobs compute the costs for the execution (with the grid's `costsPerCPUMinute`) and sum them up. Additionally, add the scheduling costs (the static costs that incurred for assigning the jobs in the first place). Use your PriceManagementBean to calculate the scheduling costs. Don't forget the user's discount, if he has a membership for a certain grid. Return the bill as a simple String. The bill should contain the total price, the price per job, the setup costs and execution costs and the number of computers that have been used per job. As soon as the bill for one job is finished you can set the payment status of this job to paid.

  As collecting this data may take some time, the method should be executed asynchronously. The client simply invokes this method, and immediately gets control back. This way, the client can continue processing while the calculation is running, and receives the collected data asynchronously when it's finished. Check out the possibilities that EJB 3.1 provides for this purpose.

1.c. Audit-Interceptor (4 Points)

- **Develop an audit interceptor and apply it to the JobManagementBean**:

  Since the JobManagementBean is essential to our system we now implement some simple logging to get some insight into the internal workings of the bean. We implement our logger as an audit interceptor. The interceptor should persist the invocation time, method name, parameters (index, class and value) and result value (or exception value in case of failure). You may simply invoke the toString() method of objects to convert results to persistable strings (but make sure to check for null values). Note that transactions are usually rolled back in case of an exception, and that this behavior would also influence our interceptor by default. Therefore, check how to bypass this behavior to be able to persist the audit even in case of a failure (e.g., if a job cannot be scheduled). Add a method to the GeneralManagementBean to retrieve all these audits (do not directly return entities, but create simple data transfer objects!).

1.d. Client application

Now the only thing left to do is write a client application to test our EJB system.

The included **appserv-rt.jar** already contains a preconfigured jndi.properties file sufficient for your needs. So calling `Context ctx = new InitialContext();` is enough, you dont have to supply any properties. JNDI names for remote beans are standardized since Java EE 6; look them up using the following pattern: `java:global[<app-name>]/<module-name>/<bean-name>#<interface-name>`, e.g. `java:global/dst2_1/TestBean`.

In your client program you have to execute the following steps (users hit <Enter> to proceed to the next step):

- Insert your test data.

- Use your GeneralManagementBean to set some prices.
- Afterwards obtain a reference to your JobAssignmentBean, try to login with invalid values, then login successfully, add some valid job assignments, request the current assigned amount of jobs and finally successfully submit your temporary job list.
- Replay the last step with a different user, but this time try to assign more jobs for a grid than there are free computers. Delete the job assignments for a grid, request the assigned amount of jobs and finally successfully submit your temporary job list.
- Wait for some time so that your jobs are finished.
- Use the GeneralManagementBean to get the bill for all finished jobs.
- Finally get all saved audits from the Audit-Interceptor.

For each step you need to provide reasonable output (so we can easily keep track of what is going on). Please test properly, in order to avoid problems when we check your assignment. However, your solution will be run only once, so you may hardcode IDs.

2. Web Services (8 Points)

While EJBs are an important technology of enterprise distributed computing, their usability as integration technology (for instance, between applications of business partners in a value chain, or between applications of different departments) is very limited. For such tasks, SOAP-based Web services have emerged as the current de facto standard. SOAP-based Web services allow heterogeneous applications (i.e., applications written in different programming languages and running on different platforms) to communicate via standard HTTP, across company (and, hence, firewall) borders.

Service and Client (4 Points)

In this task, you should develop a simple SOAP and WSDL-based service, which (anonymous) external users can invoke to find some base statistics about the utilization of grids. The service should provide one Web service operation, which takes a simple String as argument (a grid name). The service returns all finished executions of this grid using custom data transfer objects (never expose your internal data layer to external parties!). The custom DTO should contain the grid name and a list of executions, each in turn containing the start date, the end date, and the number of CPUs that was used by this execution in total. **Do not expose any other information, such as database IDs, via your Web service!**

The service should be developed using JAX-WS on top of a new EJB (select the correct type of bean for this job), the JobStatisticsBean. Use the reference implementation of JAX-WS, which is part of your Java installation as well as of the Glassfish application server. You will easily find simple examples on how to create JAX-WS services on the web. For more complex tasks, the JAX-WS specification[7] is the best place to look (scroll to the appendix, *Annotations*).

These are the technical specifications for your service:

- The endpoint of your service should be `http://localhost:8080/JobStatistics/service`, consequently, the WSDL contract of your service should be located at `http://localhost:8080/-JobStatistics/service?wsdl`.
- Your service should make use of WS-Addressing for message routing (see here[8] for an example). Assign explicit input, output and fault actions.
- The input parameter of the service (the grid name) should be transported as a header parameter (i.e., the String should go into the header of the SOAP message, instead of the body). The service response should be transported in the message body, as usual.

[7] http://download.oracle.com/otndocs/jcp/jaxws-2.2-mrel3-evalu-oth-JSpec/
[8] http://jax-ws.java.net/jax-ws-21-ea3/docs/wsaddressing.html

- If the passed grid name could not be found in your database, send back a fault message (a SOAP fault) of type UnknownGridFault.

After building and deploying your service, you should be able to access the WSDL contract of your service at the location specified above. You can test your service without writing code using the soapUI tool[9].

Now it is time to build a simple client for the service. To this extent, we first need to create client stubs from the WSDL contract. To ease this task, we provide an ant target `ws-import-client` as part of our build environment. The target assumes that the `wsimport` tool (part of Java 6) is in your path and that you have actually published your service at the endpoint specified above. The ant task will generate the stub code to `dst2.ejb.generated`. Rerunning the task will **delete** all code in this package, so do not change the generated stubs.

Use the generated stubs to invoke your service as an additional last step of your client application, and write the results to the console.

Note: depending on the Java version you use, the included JAX-WS version might be too old to use with the current version of Glassfish. If you run into troubles invoking your service, you should update JAX-WS by copying lib/jaxws-api.jar from the template to $JAVA_HOME/lib/endorsed.

A Peek Under the Cover (4 Points)

So far, we have seen two quite different remoting paradigms in action (EJB remoting and SOAP-based Web services). For this task, you need to take a look under the cover of these technologies. Use a network sniffer, for instance Wireshark[10], to save the client/server interactions for one sample EJB remote invocation and one Web service invocation. Store the sent and received messages to the folder `remoting_artifacts` in your project. Additionally, download the WSDL contract of your Web service and store it to the same directory. Study these artifacts. During the practice lessons, you should be able to discuss what is going on in some detail.

3. Dependency Injection (10 Points)

As Dependency Injection (Inversion of Control) is a very important and omnipresent feature of modern frameworks and application servers (e.g., Spring[11], EJB[12], CDI[13]), we will now implement a simple custom Dependency Injection Controller using annotations. All code for this task has to be put into the **2_injection** subfolders. The solution for this task has no relationship to the grid computing case study. Read the assignment text to the end (until the start of the theory questions before starting to code!

Task A: Standalone Injection Controller (6 Points)

Your task is to create a thread safe implementation of the supplied `dst3.depinj.IInjectionController` interface (**take a look at it before you continue reading**):

- All classes of which objects should be initialized or injected using the controller must be annotated by a `Component` annotation. It has to be possible to specify the scope of a component: `Singleton` (only one instance is created within the controller and shared between injected objects) or `Prototype` (a new instance is created every time one is requested). In case the controller is advised to initialize (i.e., the initialize() method is called by the user) an object of a singleton component it already knows an instance of, it should throw an `InjectionException`.

[9] http://www.soapui.org/
[10] http://www.wireshark.org/
[11] http://www.springsource.org/
[12] http://www.oracle.com/technetwork/java/javaee/ejb/index.html
[13] http://seamframework.org/Weld

- All Component-annotated classes must have an id, which is annotated as `ComponentId`. The id has to be unique (in the injector's scope) and of type `Long`. The id field is set by the injector if the object was successfully initialized. If no id is present or the id variable has the wrong type throw an `InjectionException`.
- Every field (inherited, public/private and so on) of a component that is annotated by `Inject` has to be processed. It has to be possible to define whether the injection is `required` (if false, no exception is raised if it is not possible to set this field) and to specify a concrete subtype that should be instantiated (`specificType`). This specific type is optional (if not present, the declared type is used for injection).
- It is not required to deal with circular dependencies. However, you have to completely initialize hierarchically composed objects and report any naming ambiguities that occur. Wrap checked exceptions in InjectionException.
- Put your code into the **injector** project.

Task B: Transparent Injection Controller (4 Points)

To use our dependency injector framework we always have to write the same lines of code to create instances and then let the controller initialize them. To remove this requirement, we are now going to use some bytecode manipulation (or code instrumentation). Use bytecode manipulation to insert a code snippet into each constructor of a `Component` annotated class in which you use an `IInjectionController` instance to initialize the object. Note that it might be necessary to modify the implementation you wrote before - it's not necessary that both execution modes work in parallel.

Study the java.lang.instrument[14] package description and implement a `ClassFileTransformer` that modifies the byte code using the Javassist[15] library (the required .jar file is already part of the template project). Read the tutorial to understand the concepts of Javassist. Put your code into the **agent** project (a jar library will be created and made available for the sample project). Complete the **dist** target to set the premain class attribute.

The following code snippet (Listing 1) illustrates the controller's functionality and how it is used in both tasks. You also have to provide a **sample application** that shows the full palette of features of your dependency injection controller. Put your code into the **sample** project. Complete the **run** or **run-with-agent target** (depending on what you have solved).

[14] http://java.sun.com/javase/6/docs/api/java/lang/instrument/package-summary.html
[15] http://www.csg.is.titech.ac.jp/ chiba/javassist/

```
@Component(scope = ScopeType.PROTOTYPE)
public class ControllerWithInjections {

     @ComponentId
     private Long id;

     @Inject(specificType = SimpleInterfaceImpl.class)
     private SimpleInterface si;

     public void callSi() {
          si.fooBar();
     }

     public static void taskA() {
          IInjectionController ic = ...;
          ControllerWithInjections cwi =
               new ControllerWithInjections();
          ic.initialize(cwi);
          cwi.callSi(); // output expected
     }

     public static void taskB() {
          ControllerWithInjections cwi =
               new ControllerWithInjections();
          cwi.callSi(); // output expected
     }
}

public interface SimpleInterface {
     void fooBar();
}

@Component(scope = ScopeType.SINGLETON)
public class SimpleInterfaceImpl implements SimpleInterface {

     @ComponentId
     private Long id;

     public void fooBar() {
          System.out.println("[SimpleIntefaceImpl] id: " +
               id + " fooBar called!");
     }
}
```

Listing 1: Code Snippet

B. Theory Part

The following questions will be discussed during the practice lesson. At the beginning of the each lesson we hand out a list where you can specify which questions you have prepared and are willing to present. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a correct and well-founded answer, you will lose **all** points for the theory part of this assignment.

4. EJB Lifecycles (1 Points)

Explain the lifecycle of each bean type defined in the EJB 3.1 specification. What optimizations can the EJB container perform for the respective type? Also think about typical use cases the respective bean type provides to the developer.

5. Dependency Injection (2 Points)

Explain the way dependency injection is performed by the EJB container. What kind of resources may be injected into a bean, and what are the different annotations that can be used?

6. Java Transaction API (2 Points)

The EJB architecture provides a mechanism for distributed transactions. Explain the two ways how transactions can be defined. How is the concept of distributed transactions accomplished behind the scenes, i.e. what tasks have to be performed by the EJB container?

8. Remoting Technologies (1 Points)

Compare EJB remoting and Web services. When would you use one technology, and when the other? Is one of them strictly superior? How do these technologies relate to other remoting technologies that you might know from other lectures (for instance, Java RMI, CORBA, or even socket programming)?